ESTRUTURAS DE DADOS E ALGORITMOS
Departamento de Engenharia Eletrotécnica e de Computadores
Faculdade de Ciências e Tecnologia da Universidade de Coimbra

FICHA 7
TÓPICOS AVANÇADOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

7.1 Objetivos

Objetivos que o aluno deve atingir com esta ficha:
- Compreender o conceito de membros constantes;
- Compreender o conceito de derivação de classes, heranças e hierarquia de classes;
- Compreender o conceito de membro virtual e polimorfismo;
- Compreender o conceito de sobrecarga de operadores.

7.2 Membros constantes

Quando um objeto de uma classe é declarado como constante (`const MyClass myObject;`) os seus atributos não podem ser alterados (só podem ser lidos). No entanto, o construtor ainda é chamado e é-lhe permitido inicializar e modificar os atributos.

Os objetos constantes só podem invocar métodos declarados como constantes. Um método declarado como constante não pode alterar os atributos não estáticos dos objetos da classe (pode alterar atributos estáticos). Também não pode chamar métodos não constantes. Para declarar um método como constante usa-se a palavra-chave `const` depois do parêntesis da direita da lista de parâmetros, antes da chaveta de início do corpo do método:

```
int getLargura() const {return largura;}
```

Os atributos declarados como constantes não podem ser alterados depois da criação do objeto. Podem ser inicializados após a lista de parâmetros, antes do corpo do construtor:

```
class CPoligono {
        int largura, altura;
        const int numLados; // atributo constante
  public:
        CPoligono(int larg, int alt, int nLad) : numLados(nLad)
                    { largura = larg; altura = alt; }
        int getLargura(void) const;      // método constante
  //    void setLargura(int larg) const { largura = larg; } // Erro!!!
}; // Fim da classe
//-----------------------------------------------------------------
int CPoligono::getLargura(void) const { return largura; }
//-----------------------------------------------------------------
```

7.3 Derivação de classes, membros virtuais e polimorfismo

Uma das grandes vantagens da programação orientada a objetos reside na possibilidade de facilmente podermos reutilizar código escrito anteriormente. Uma noção que contribui para esse facto é a noção de ***herança*** de classes. É possível criar novas *classes derivadas* que sejam descendentes de uma *classe base* já existente. Uma classe derivada inclui os novos membros que nela sejam definidos e também todos os membros (públicos, protegidos e privados) da classe base, com a exceção dos construtores e do destrutor, da sobrecarga de operadores e da definição das classes e funções amigas. Embora os construtores e os destrutores não sejam herdados, a classe derivada mantém os construtores e o destrutor da classe base, que são sempre chamados antes de ser chamado um construtor ou o destrutor da classe derivada.

Imagine que pretendia criar uma nova classe que representasse retângulos a cheio ou blocos. Em vez de criarmos uma classe completamente nova, onde se teria de redefinir todos os atributos e métodos que definem um retângulo, poderíamos aproveitar a definição da classe `CRetangulo` anteriormente definida (ver ficha anterior):

```
class CRetangulo {
  protected:    // estes membros são visíveis pelas classes derivadas
     // Atributos do retângulo
     int altura;
     int comprimento;

  public:
     CRetangulo();                        // Construtor por defeito
     CRetangulo(int a, int l);            // Construtor por parâmetros
     CRetangulo(const CRetangulo &r);     // Construtor por cópia
     void desenha(void) const;            // Método que desenha o retângulo
}; // fim da classe
//------------------------------------------------------------------------
```

Podemos aproveitar a definição desta classe para definir uma nova classe `CBloco` como uma *classe derivada* de `CRetangulo`. Note que como pretendemos também aceder nos métodos da classe derivada aos membros `altura` e `comprimento` herdados da classe `CRetangulo`, tivemos que alterar a visibilidade destes membros na classe `CRetangulo` de privados para protegidos já que os membros privados não são visíveis por classes derivadas (exceto se forem amigas).

```
class CBloco : public CRetangulo {
  protected:
    bool cheio;

  public:
     /* Temos de definir os construtores desta nova classe, uma vez que
         eles não são herdados da classe base */

     CBloco() {
        altura = comprimento = 0;
        cheio = true;
```

```
        }

        CBloco(int a, int c, bool ch = true) {
           altura = a;
           comprimento = C;
           cheio = ch;
        }

        void inicializaCheio(bool ch) {
           cheio = ch;
        }

        void desenha(void) const;

    }; // fim da declaração da classe
    //-----------------------------------------------------------------
    void CBloco::desenha(void) const {
        cout << endl;
        if (cheio) {
           for (int i=0; i < altura; i++) {
              for (int j=0; j < comprimento; j++)  cout << "*";
              cout << endl;
           }
           cout << endl;
        } else
        CRetangulo::desenha();  // Chama o método da classe base
    }
    //-----------------------------------------------------------------
```

Repare que nesta nova classe redefinimos o comportamento do método `desenha()`. Neste caso, podemos desenhar o retângulo a cheio ou não, dependendo do valor do membro `cheio`. No caso de `cheio` ser `false` então o desenho do retângulo é igual ao da classe base e chamamos o método `desenha()` da classe `CRetangulo`.

Para testarmos esta nova classe, podemos definir o programa principal apresentado a seguir:

```
    int main() {
       CBloco *umBloco = new CBloco(7,5, true);

       cout << "Agora e' cheio..." << endl;
       umBloco->desenha();

       umBloco->inicializaCheio(false);
       cout << "Agora ja' nao..." << endl;
       umBloco->desenha();

       delete umBloco;
       return 0;
```

```
}
```

Por forma a compatibilizar a utilização de ponteiros do tipo da classe base para referenciar as suas classes derivadas, por vezes é necessário declarar na classe base métodos que apenas vão ser definidos nas classes derivadas. Esses membros são definidos recorrendo à palavra reservada `virtual`.

Os métodos virtuais devem ser redefinidos nas classes derivadas. Se tal não acontecer é aplicado o comportamento definido na classe base. Esta propriedade designa-se por polimorfismo porque cada classe derivada pode alterar a forma de funcionamento dos métodos virtuais.

Para ilustrar este conceito tomemos o seguinte exemplo: consideremos uma nova classe, derivada da classe `CBloco`, e que produz blocos "redondos" cujos cantos do perímetro não são preenchidos.

```
class CRetangulo{
   ...
   virtual void desenha(void) const; /* Permite referenciar este
   ...                                  método nas classes derivadas */
}; // fim da declaração da classe
//---------------------------------------------------------------------
class CBlocoRedondo : public CBloco {
public:
   void desenha(void) const;
   CBlocoRedondo(){ }
   CBlocoRedondo(int a, int c, bool ch = true) {
     altura=a;
     comprimento=c;
     cheio=ch;
   }
   ~CBlocoRedondo(){}
}; // fim da declaração da classe
//---------------------------------------------------------------------
void CBlocoRedondo::desenha() const {
   cout << endl;
   for (int i = 0; i < altura; i++) {
     for (int j = 0; j < comprimento; j++) {
       if ((i > 0 && i < altura - 1 && j > 0 &&
            j < comprimento - 1 && !cheio) ||
           (i == 0 && j == 0) || (i == altura - 1 && j == 0) ||
           (i == 0 && j == comprimento - 1) ||
           (i == altura - 1 && j == comprimento - 1))
         cout << " ";
       else
         cout << "*";
     }
     cout << endl;
   }
   cout << endl;
}
//---------------------------------------------------------------------
```

Uma vez que a figura é diferente das anteriores, o método `desenha()` foi alterado para permitir desenhar esta nova figura. As três classes podem ser manipuladas da seguinte forma:

```
int main(void) {

   CBloco *umBlocoRedondo = new CBlocoRedondo(5, 5, true);
   cout << "Agora e' cheio..." << endl;
   umBlocoRedondo->desenha();

   umBlocoRedondo->inicializaCheio(false);
   cout << "Agora ja' nao..." << endl;
   umBlocoRedondo->desenha();
   delete umBlocoRedondo;

   CRetangulo *pFigura = new CBloco(3, 4);
   // Será que o programa sabe qual dos métodos desenha() chamar?
   pFigura->desenha();
   delete pFigura;

   // E aqui?
   pFigura = new CBlocoRedondo(4, 4);
   pFigura->desenha();
   delete pFigura;

   pFigura = new CRetangulo(3, 4);
   // E desta, será mais fácil de ver o que desenha?
   pFigura->desenha();
   delete pFigura;
   return 0;
}
```

Consegue "ver" as vantagens na utilização deste conceito em funções que lidam com parâmetros que são retângulos em geral?

7.4 Sobrecarga de operadores e construtores por cópia e conversão – funções amigas

Um dos conceitos mais úteis em programação orientada a objetos é o conceito de *sobrecarga de operadores*. Para além da sobrecarga de funções, é também possível modificar o comportamento de grande parte dos operadores, permitindo que estes se adaptem ao contexto de cada classe. Os operadores passíveis de serem sobrecarregados para classes definidas pelo programador (e não para os tipos convencionais) são:

```
+     -     *     /     %     ^     &     |     ~     !     =     <     >
+=    -=    *=    /=    %=    ^=    &=    |=    <<    >>    >>=   <<=   = =
!=    <=    >=    &&    ||    ++    --    ,     ->*   ->    ()    []
new   delete
```

Para ilustrar o conceito de sobrecarga de operadores, iremos de seguida sobrecarregar o operador binário + da nossa classe `CRetangulo`. O objetivo é somar dois retângulos, somando os lados, ou então somar um retângulo com uma constante, adicionando essa constante aos lados. Temos de ter presente que, em termos de classes, qualquer expressão do tipo `objeto1 + objeto2` é substituída pelo método `objeto1.operator + (objeto2)`. A partir daqui, fica claro que o primeiro operando é o próprio objeto onde se efetua a sobrecarga da operação e o segundo operando é passado como parâmetro do método. As duas formas de sobrecarga do operador + referidas anteriormente podem ser definidas da seguinte forma:

```
public:
   CRetangulo operator+(const CRetangulo &ret);
   CRetangulo operator+(const int aumento);
   ...
}; // fim da classe
//-----------------------------------------------------------------
/* Implementação da sobrecarga do operador + por forma a permitir a
   soma de dois objetos do tipo CRetangulo */
CRetangulo CRetangulo::operator + (const CRetangulo &ret) {
    return CRetangulo (altura+ret.altura, comprimento+ret.comprimento);
}
//-----------------------------------------------------------------
/* Implementação da sobrecarga do operador + por forma a
   permitir adicionar um retângulo a um inteiro */
CRetangulo CRetangulo::operator + (const int aumento) {
    return CRetangulo(altura + aumento, comprimento + aumento);
}
```

Repare na utilização da declaração `const` nos parâmetros de entrada dos dois métodos apresentados. A ideia é impedir que um parâmetro, mesmo que passado por referência, seja alterado quando passado à função. E porquê a passagem por referência neste caso e não a passagem de parâmetros por valor? A resposta tem a ver com uma questão de eficiência: é menos eficiente fazer uma cópia desse objeto, principalmente de um objeto que ocupe muita memória, do que utilizar um "pseudónimo", i.e. uma referência para o designar.

E se quisermos que a última sobrecarga, que permite adicionar uma constante inteira a ambos os lados do retângulo, possa, além do caso `CRetangulo + int`, também estender-se ao caso `int + CRetangulo`? Teremos de recorrer a uma estratégia diferente, uma vez que a sobrecarga de operadores numa classe assume sempre que o objeto atual é o primeiro parâmetro. A solução é definir uma função *global* da seguinte forma:

```
/* Esta função global funciona da seguinte forma:
   CRetangulo operator+(int operando1, CRetangulo operando2)
   operando1 (aumento) é o operando da esquerda e é do tipo inteiro e
   operando2 (ret) é o da direita e é do tipo CRetangulo. */
CRetangulo operator + (const int aumento, const CRetangulo &ret) {
   return CRetangulo (ret.altura + aumento, ret.comprimento + aumento);
}
```

Esta função tem, porém, um problema: ela tenta aceder a membros privados de `CRetangulo` (`altura` e `comprimento`). Como sabe, não é possível aceder a membros privados fora do âmbito da classe...

Para resolver esta questão foi criado um tipo de declaração especial: a declaração `friend`. Esta declaração define o conceito de funções amigas de uma classe. Uma *função amiga* de uma classe tem permissão para aceder aos membros privados dessa classe. No exemplo apresentado acima teremos de adicionar à declaração da classe `CRetangulo` a linha que se segue:

```
    public:
       ...
       friend CRetangulo operator+(int aumento, const CRetangulo &ret);
```

A partir do momento que esta função global é declarada "amiga", ela passa a poder aceder sem reservas a todos os membros da classe `CRetangulo`.

A seguir apresenta-se um exemplo de utilização das sobrecargas do operador +:

```
    int main() {
       cout << "Agora demonstremos o operador +..." << endl;
       CRetangulo ret1(1,1);
       ret1 = ret1 + CRetangulo(1,2);
       ret1.desenha();
       ret1 = ret1 + 1;
       ret1.desenha();
       ret1 = 2 + Ret1;
       ret1.desenha();
    }
```

Para saber mais sobre os operadores em C++, ver os links:
http://www.cplusplus.com/doc/tutorial/operators/
http://en.wikipedia.org/wiki/Operators_in_C_and_C++
http://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

7.5 Exercícios sugeridos

Com o conjunto de exercícios apresentados a seguir, pretende-se que o aluno consolide os seus conhecimentos sobre tópicos avançados de programação orientada a objetos. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios que não seja possível realizar durante as aulas práticas.

Problema 7.1 – Fácil

Utilizando a classe `CData` definida na ficha anterior, derive uma nova classe `CFeriado` que permita representar os feriados no calendário. Esta nova classe, para além da data, deve também armazenar a descrição do feriado. Altere o método `escreveData()` na nova classe por forma a permitir escrever a data seguida da descrição do feriado (por exemplo: 17:53 de 10/06/2019 – Dia de Portugal).

```
class CData{
  private:     // deverá ser "protected" para ser visível da classe derivada
    int dia;
```

```
    int mes;
    int ano;
    int horas;
    int minutos;
  public:
    CData();                      // Construtor por defeito (00:00 de 01.01.1970)
    CData(int d, int m, int a, int h, int min);     // Construtor por enumeração
    CData(const CData &data);     // Construtor por cópia
    int devolveDia()const;        // Devolve o dia da data atual
    int devolveMes()const;        // Devolve o mês da data atual
    int devolveAno()const;        // Devolve o ano da data atual
    void mudaHoraDoDia(int, int);// Permite alterar a hora do dia
    void mudaData(int, int, int);// Permite introduzir uma nova data
     void escreveData()const;     // Escreve hora e data no ecrã no
                                  //  formato hh:mm de dd/mm/aaaa
    int numeroFDS()const;         // Devolve o número de fins de semana decorridos
                                  //  desde o início do ano
}; // fim da declaração da classe
```

Implemente a sobrecarga do operador – por forma a devolver o número de dias que separam dois feriados diferentes.

Problema 7.2 – Fácil

Dada a classe `CCash` definida na ficha anterior, que permite definir um preço em euros, implemente as seguintes sobrecargas de operadores:

- Operador `+` – permite somar um objeto do tipo `CCash` a uma constante do mesmo tipo;
- Operador `!=` – permite verificar se dois preços são diferentes);
- Operador `%` – devolve um `int` que é a soma de apenas as componentes cêntimos de um objeto do tipo `CCash` com uma constante do mesmo tipo.

Problema 7.3 – Médio

Considere a classe `CPolinomio` que pretende representar polinómios do segundo grau do tipo $a_2x^2+a_1x+a_0$:

```
class CPolinomio{
   private:
     int grau;          // Contém sempre o grau do polinómio.
     double coef[3];    // Coeficientes do polinómio: [0]=a0, [1]=a1,…
   public:
     CPolinomio ();    // Inicialização por defeito (polinómio = 1)
     CPolinomio(double a2, double a1, double a0); /* construtor por parâmetros:
                                                     atualiza grau */
     CPolinomio(const CPolinomio &p);           // Construtor por cópia
     ~CPolinomio( );
}; // fim da classe
```

Implemente um método `escreve()` que permita escrever o polinómio representado pelo objeto. Implemente também a sobrecarga dos seguintes operadores: `-` (subtração de dois polinómios), `*`

(multiplicação dos coeficientes por uma constante inteira), e finalmente o operador `[]` que permita devolver o coeficiente correspondente ao índice passado como parâmetro (por exemplo, `polinomio[2]` deverá devolver o coeficiente a_2).

Problema 7.4 – Médio

Implemente uma classe `CVetor` que defina vetores tridimensionais $[x\ y\ z]^T$ através das suas coordenadas. Implemente os seguintes métodos:

- `modulo()` – devolve o módulo do vetor;
- `inverte()` – vetor passa a ter sentido contrário (simétrico);
- Sobrecarga do operador `+` – implementa a soma de dois vetores);
- Sobrecarga do operador `!=` – neste caso, considera-se que dois vetores são diferentes se têm módulos diferentes).

Problema 7.5 – Médio/Trabalhoso

Implemente a seguinte classe denominada `CEsfera`:

```
class CEsfera{
   double raio; // Raio da Esfera
   char cor[15];// String que indica a cor da Esfera

public:
   CEsfera();                // Construtor por defeito (Raio = 1.0, Cor = Branco)
   CEsfera(double r, const char* c);      // Construtor por parâmetros
   CEsfera(const CEsfera& e);// Construtor por cópia
   ~CEsfera();               // Destrutor
   double volume() const;    // Devolve o volume da Esfera(4./3*pi*Raio^3)

  // Sobrecarga de operadores:
   CEsfera operator+(const CEsfera&) const; // Sobrecarga do operador +
                      // somam-se os raios e a cor é a da primeira Esfera
   CEsfera& operator=(const CEsfera&);    // Atribuição de uma Esfera a outra
                                          //  Esfera
   bool operator==(const CEsfera& e) const; //Testa igualdade entre esferas
   bool operator==(double r) const; // Idem em que r é o raio da segunda
}; // fim da classe
```

Problema 7.6 – Médio/Trabalhoso

Implemente a seguinte classe denominada `CStreamVideo`:

```
class CStreamVideo {
  private:
    char stream[1024/8];  // Armazena os até 1024 bits da sequência de video
    int numBits;          // nº de bits da sequência (múltiplo de 8)
    bool erro;            // Este valor, que deve ser sempre atualizado, é igual a
       // true se o resultado da operação XOR entre todos os bits for 1
    bool verificaErro();  // permite atualizar o valor do membro Erro

  public:
    CStreamVideo();        // Inicialização por defeito com tudo a zero
    CStreamVideo(const char *seq, int nBits); /* Inicialização através de...
                   ... uma tabela de 0's e 1's (se não for múltiplo de 8 trunca) */
    CStreamVideo(const CStreamVideo &sv);     // Inicialização por cópia
    ~CStreamVideo( );                         //Destrutor
    bool Erro() const;                        // retorna o valor do membro Erro
    CStreamVideo operator + (const CStreamVideo &sv); // Concatena Streams
}; // fim da declaração da classe
```

Problema 7.7 – Difícil/Trabalhoso

Implemente uma classe denominada `CString` com a seguinte definição:

```
    class CString{
       int Comprimento;          // Comprimento da String
       char* PonString;          /* Ponteiro para a zona de memória onde a String
                                     está armazenada */
    public:
       CString();                // Construtor por defeito -> String vazia
       CString(const char *);    // Inicializa string a partir de um conjunto
                                 // de caracteres
       CString(const String&);   // Construtor por cópia
       ~CString();               // Destrutor
       GetLine();          // Pede uma linha ao utilizador e guarda-a na string
       int Length();// Devolve o comprimento da String

      // Sobrecarga de operadores:
       CString& operator=(const char*); // Atribuicao de um conjunto de
                                        //  carateres a uma String
       CString& operator=(const CString&);    // Atribuicao de uma String a outra
                                              // String

// continua...
```

```
    CString operator()(int,int);     // Devolve uma Substring, dada a posição
                                     //  inicial e o comprimento
    char& operator[](int);           // Devolve o caracter numa dada posicao
    CString& operator+=(const CString&);   // concatenacao de Strings, por
                                           // auto-incremento
    bool operator==(const CString&);       // Testa igualdade entre Strings
    bool operator==(const char*);          // Testa igualdade entre uma String
                                           //  e uma cadeia de carateres
    bool operator!();                // Testa se String nula?
    bool operator<(CString &);// Operador <, para comparação de Strings
    bool operator>(CString &);// Operador >, para comparação de Strings
  }; // fim da declaração da classe
```

Problema 7.8 – Fácil **(adaptado do teste de frequência de 23/06/2015)**

Considere a classe de objetos `CEsfera` para representar esferas.

```
class CEsfera{            // representa uma esfera de centro (x0,y0,z0) e raio r
  protected:
      double centro[3];
      double raio;
  public:
      CEsfera() { ... }   // construtor por omissão
      CEsfera(double x, double y, double z, double r) { ... } // por enumeração
      void mostra() { for (int i=0; i<3; i++) cout << centro[i] << ", ";
                  cout << raio; }
};
```

a) Declare uma classe derivada `CBola` para incluir o atributo `peso` (do tipo `double`).
b) Na classe derivada (re)defina o método `mostra()` para que apresente no ecrã todos os atributos, incluindo os herdados da classe base.
c) Defina um método que devolva o valor da densidade de uma bola (=peso/volume, volume=4/3 π r^3). Este método não deve escrever nada no ecrã.

Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.9 – Médio **(adaptado do teste de frequência de 06/05/2015)**

Considere a classe de objetos `CTempo` para representar tempos.

```
class CTempo {
      int horas, minutos, segundos;
public:
      CTempo() { horas = minutos = segundos = 0; }
      CTempo(float t) { set(t); }
      void set(float t);  // ex: se t = 2.56, então horas=2, minutos=33, segundos=36
      float get(void) const; // ex: se horas=2, minutos=33, segundos=36, devolve 2.56
      void mostra(void) const { cout << setfill('0') << setw(2) << horas << ':' <<
                          setw(2) << minutos<<':'<< setw(2) << segundos << endl; }
      CTempo operator -(const CTempo& t2);
};
```

a) Defina o método constante `float CTempo::get(void) const`.
Sugestão: use a fórmula `t = horas + minutos/60.0 + segundos/3600.0`.
b) Defina o método `void CTempo::set(float t)` com base no algoritmo descrito a seguir em pseudo-código.

```
horas = parte inteira de t
t = (t – horas) * 60
minutos = parte inteira de t
segundos = arredonda ((t – minutos) * 60) para o inteiro mais próximo
```

c) Defina o método `CTempo CTempo::operator -(const CTempo& t2)`.
Sugestão: chame o método `get()` para ambos os objetos e use o método `set()` no novo objeto.

Implemente uma função `main()` para ir testando a implementação da classe.

Problema 7.10 – Médio **(adaptado do exame de recurso de 10/07/2015)**

Considere a classe de objetos `CPiramide` para representar pirâmides regulares. Uma pirâmide é regular se a base for um polígono regular, ou seja um polígono cujos lados tenham todos o mesmo comprimento. Um apótema da base é um segmento de reta que liga o centro do polígono da base ao ponto médio de um dos seus lados.

```
class CPiramide{
  protected:
      int numLadosBase;          // número de lados da base regular
      double altura, lado;       // altura da pirâmide e comprimento dos lados da base
      double apotema;            // comprimento do apótema da base
  public:
      CPiramide() { numLadosBase = 3; altura = 1.0; lado = 1.0; setApotema(); }
      CPiramide(int n, double a, double l) { ... }
      void mostra()const{ cout << “Pirâmide regular de altura ” << altura <<
                          “ e base de lado ” << lado << endl; }
      void setApotema() { apotema = lado / (2.0 * tan(3.14159265 / numLadosBase)); }
      double areaBase() const { return (0.5 * apotema * numLadosBase * lado); }
      virtual double volume() = 0;     // método para calcular o volume da pirâmide
};
```

a) Declare uma classe derivada `CPiramideHexagonal` para representar pirâmides retas regulares cuja base é um hexágono regular. Uma pirâmide é reta se a projeção do vértice da pirâmide na base coincide com o centro geométrico da base. Declare e implemente nesta classe derivada um construtor que aceita dois parâmetros: a altura da pirâmide e o comprimento dos lados da base. Este construtor inicializa todos os atributos do objeto.
b) Defina na classe derivada o método virtual `volume()`.
Nota: O volume da pirâmide é dado pela expressão volume = 1/3 x Área da Base x Altura.
Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.11 – Médio **(adaptado do teste de frequência de 26/04/2017)**

Considere uma aplicação informática que tem por objetivo gerir dados de pessoas. Nesta aplicação, existe uma classe de objetos `CPessoas` que permite definir um conjunto de pessoas. Possui os atributos (privados) `npessoas` (`int`) e uma tabela de nomes (`string`) e outra de datas de nascimento (`int`) em que a data é expressa no formato aaaammdd. As tabelas têm dimensão `MAX` (definida como constante inteira no início

do programa e igual a 100). Existem os métodos (públicos) seguintes: i) um construtor por defeito/omissão (tabelas vazias); ii) um construtor por cópia; iii) um método para inserir uma pessoa (dados o nome e a data de nascimento como parâmetros); iv) um método `ordena()` que ordena as pessoas por data de nascimento (altera ambas as tabelas).

a) Escreva a declaração da classe de objetos `CPessoas`. Nesta alínea, não deve implementar qq. método.
b) Implemente os construtores por defeito, por cópia e o método para inserir.
c) Implemente o método `ordena()`.

Considere agora que nessa aplicação informática foi criada uma classe derivada da anterior, `CAdultos`, que inclui mais um atributo (privado) inteiro (`nadultos`, o nº de pessoas com mais de 18 anos). Esta classe deve ter acesso a todos os atributos da classe base. Existem os métodos (públicos) da classe base e ainda os métodos `getNumAdultos()` (devolve o valor do atributo `nadultos`) e o método `atualiza()` que ordena as tabelas e atualiza o atributo `nadultos`, usando a data de hoje como referência (considere por exemplo que a data de hoje é 20200125 ou outra qualquer à sua escolha).

d) Re-escreva a declaração da classe de objetos `CPessoas` tendo em conta a nova classe derivada.
e) Escreva a declaração da classe de objetos `CAdultos`. Nesta alínea, não deve implementar qq. método.
f) Implemente o método `atualiza()`. Deve ter em conta que as tabelas são primeiro ordenadas (chamando o método herdado `ordena()`) antes de contar o nº de adultos (i.e., não deve pesquisar toda a tabela).

Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.12 – Médio **(adaptado do exame de recurso de 03/07/2017)**

A classe de objetos `CAngulo` permite definir um ângulo em graus, minutos e segundos. Possui os atributos privados `graus`, `minutos` e `segundos` (todos do tipo `int`) e os métodos seguintes: i) um construtor por defeito (objeto com 10º5'15''); ii) um construtor por enumeração (recebe 3 inteiros, correspondentes aos graus, minutos e segundos); iii) um construtor por cópia; iv) a sobrecarga do operador `<`.

a) Escreva a declaração da classe de objetos `CAngulo`. Nesta alínea, não deve implementar qualquer método.
b) Implemente os construtores por defeito, por enumeração e por cópia da classe `CAngulo`.
c) Implemente a sobrecarga do operador `<`.

Considere agora que foi criada uma classe derivada da classe `CAngulo`, `CSetor`, que inclui mais um atributo privado inteiro: `raio`, o raio do setor circular. A classe derivada deve ter acesso a todos os atributos da classe base. Existem os métodos públicos da classe mãe e ainda o método `menorTamanhoQue()` – um setor circular é menor que outro se simultaneamente tiver menor ângulo e menor raio.

d) Re-escreva a declaração da classe de objetos `CAngulo` tendo em conta a nova classe derivada.
e) Escreva a declaração da classe de objetos `CSetor`. Nesta alínea, não deve implementar qualquer método.
f) Implemente o método `menorTamanhoQue()`. Deve chamar o operador herdado `<`.

Implemente uma função `main()` para ir testando a implementação das classes.

Problema 7.13 – Médio **(saiu no teste de frequência de 16/05/2018)**

A classe de objetos `ArrayDinamico` permite manipular uma tabela alocada dinamicamente na *heap*, do tipo `float`, de tamanho máximo igual a `tamanho` (tipo `int`), que é um múltiplo de 100 que pode ser aumentado dinamicamente se houver necessidade de inserir mais elementos do que o tamanho inicialmente previsto. Para além do atributo `tamanho`, a classe possui o atributo `array` (ponteiro para `float`) que guarda o endereço do 1.º elemento da tabela e o atributo `n` (tipo `int`) que guarda o número de elementos realmente ocupados na tabela (é sempre ≤ que tamanho). Possui pelo menos os métodos seguintes: i) construtor por defeito/omissão (tabela com `tamanho` = 100 e `n` = 0); ii) construtor que aceita um inteiro `tam` (tabela com tamanho igual ao menor múltiplo de 100 maior ou igual a tam); iii) destrutor; iv) operador `<<` que insere um novo número na posição `n` da tabela e a seguir incrementa `n` – se antes da inserção tiver sido atingido o tamanho máximo (i.e. `n==tamanho`), a tabela é redimensionada para aumentar em 100 elementos `tamanho` (aloca novo bloco de memória com `tamanho=tamanho+100` elementos, copia para o novo bloco a informação atual, atualiza o ponteiro `array` e liberta da *heap* o bloco anteriormente utilizado); v) operador `+` que adiciona dois objetos da classe, realizando a adição elemento a elemento das tabelas, sem alterar as tabelas, e devolve um objeto da classe com o resultado, de tamanho igual ao menor dos tamanhos das duas tabelas a adicionar (sendo `n` também igual ao menor `n` das duas tabelas).

a) Escreva a declaração da classe de objetos `ArrayDinamico`. Nesta alínea, não deve definir/implementar qualquer método *inline*; os métodos devem ser apenas declarados nesta alínea.
b) Defina/implemente o operador `<<`. O operador deve permitir executar a operação em cascata, ex. `a << 1.5 << 2.5`, em que a é um objeto da classe `ArrayDinamico`.
c) Defina/implemente o operador `+` da classe `ArrayDinamico`, bem como quaisquer outros métodos da classe que considere necessários para o operador poder executar corretamente.

Implemente uma função `main()` para ir testando a implementação da classe.